2016-09

A multi-threaded cryptographic pseudorandom number generator test suite

Zhang, Zhibin

Monterey, California: Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A MULTI-THREADED CRYPTOGRAPHIC PSEUDORANDOM NUMBER GENERATOR TEST SUITE

by

Zhibin Zhang

September 2016

Thesis Advisor:	Mark Gondree
Second Reader:	Paul C. Clark

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	09-23-2016	Master's Thesis 03-01-2016 to 09-23-2016

4. TITLE AND SUBTITLE

A MULTI-THREADED CRYPTOGRAPHIC PSEUDORANDOM NUMBER GENERATOR TEST SUITE

5. FUNDING NUMBERS

6. AUTHOR(S)

Zhibin Zhang

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

There are multiple applications for pseudorandom number generators, notably in simulation and cryptography. A bad pseudorandom number generator can cause misleading results in simulations or loss of security and attacks against implementations of cryptographic systems with low-entropy sequences. Pseudorandom number generator test suites provide insight and metrics for security-critical system components. This thesis added multi-threading to an existing test-suite, known as Dieharder, to significantly speed up pseudorandom number generator testing on multi-core systems. Evaluations were conducted on the original Dieharder, a threaded version of Dieharder using a POSIX-compliant thread pool (Dieharder-T), and a threaded version of Dieharder-T using OpenMP with static and dynamic scheduling. The results show that Dieharder-T with OpenMP, two threads and static scheduling completes in about half the time of the single-threaded Dieharder-T. The run-time is not halved again when the number of threads is increased to four, due to inefficient scheduling of tasks to threads. A hybrid scheduling solution is proposed to improve the performance of the multi-threaded pseudorandom number generator test suite.

14. SUBJECT TERMS

pseudorandom number generator, statistical test suite, multi-threading, Dieharder

15. NUMBER OF PAGES 71

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

A MULTI-THREADED CRYPTOGRAPHIC PSEUDORANDOM NUMBER GENERATOR TEST SUITE

Zhibin Zhang
Civilian, DSO National Laboratories, Singapore
B.Eng., Nanyang Technological University, 2009

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2016

Approved by: Mark Gondree
Thesis Advisor

Paul C. Clark
Second Reader

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

There are multiple applications for pseudorandom number generators, notably in simulation and cryptography. A bad pseudorandom number generator can cause misleading results in simulations or loss of security and attacks against implementations of cryptographic systems with low-entropy sequences. Pseudorandom number generator test suites provide insight and metrics for security-critical system components. This thesis added multi-threading to an existing test-suite, known as Dieharder, to significantly speed up pseudorandom number generator testing on multi-core systems. Evaluations were conducted on the original Dieharder, a threaded version of Dieharder using a POSIX-compliant thread pool (Dieharder-T), and a threaded version of Dieharder-T using OpenMP with static and dynamic scheduling. The results show that Dieharder-T with OpenMP, two threads and static scheduling completes in about half the time of the single-threaded Dieharder-T. The run-time is not halved again when the number of threads is increased to four, due to inefficient scheduling of tasks to threads. A hybrid scheduling solution is proposed to improve the performance of the multi-threaded pseudorandom number generator test suite.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

List of Acronyms and Abbreviations

AWS	Amazon Web Services
CPU	central processing unit
CSPRNG	cryptographically secure pseudorandom number generator
DRBG	deterministic random bit generator
EBS	Elastic Block Store
EC2	Elastic Compute Cloud
GSL	GNU scientific library
KSTEST	Kolmogorov-Smirnov test
NIST	National Institute for Standards and Technology
OpenMP	Open Multi-Processing
OTP	one-time pad
POSIX	Portable Operating System Interface
PRNG	pseudorandom number generator
RA	resolve ambiguity
RNG	random number generator
SSH	secure shell
STS	statistical test suite
TB	terabytes
TTD	test to destruction

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I would like to thank my company, DSO National Laboratories, for granting me the privilege to further my education at the Naval Postgraduate School. I would also like to express my gratitude to my thesis advisor, Professor Mark Gondree, and second reader, Paul Clark, for their guidance in completing this thesis.

I would also like to thank my wife, Priscilla, for taking such great care of me. She made it possible for me to concentrate on completing my thesis and not have to worry about anything else.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

There have been recent high-profile cases of weakness in pseudorandom number generators (PRNGs) leading to cryptographic flaws, resulting in security risks. One example is the compromise of group chats of *Cryptocat* due to a programming flaw [1]. A bug in the BigInt library caused the random number generator to have a slight bias when generating keys, making it possible to crack these keys within a day [2]. Another example is a flaw with OpenSSL's entropy pool on Debian, reducing its strength from 256 bits to 15 bits [3]. It resulted in generating only 32,767 possible secure shell (SSH) keys of a given type and size, allowing brute force to be a practical attack on the key. More recently, improper initialization of the PRNG led to android digital wallets being hijacked [4].

For military systems where PRNG design and evaluation cannot be conducted openly, testing PRNGs empirically may be the only option available for assessing the properties of these security-critical components; this is true for the many embedded and proprietary systems employed in national security applications.

PRNG test suites provide insight and metrics for these security-critical system components. However, to date there have been no good performance analyses of PRNG test suites. Review of existing statistical test suites showed that they employed a battery of efficiently implemented tests, utilizing heavy performance optimization, but these tests were run in serial. PRNGs and existing statistical test suites are reviewed in Chapter 2.

This thesis added multi-threading to *Dieharder*, a well-known PRNG test suite, to significantly speed up PRNG testing on multi-core systems. The modifications to *Dieharder* and the platform used to conduct the experiments are described in Chapter 3. To implement multi-threading, two different libraries for threading were evaluated. The first implementation used a Portable Operating System Interface (POSIX) thread pool library implemented by Mark Gondree [5], and the second implementation used Open Multi-Processing (OpenMP) [6]. The experiments with these various approaches to improve performance, and their results, are discussed in Chapter 4.

The primary contributions of this thesis are as follows:

- We modify *Dieharder* to support multi-threading, to create a variant test-suite (*Dieharder-T*).
- We integrate Gondree's thread pool library into *Dieharder-T* and evaluate performance of the test suite.
- We integrate OpenMP into *Dieharder-T* and and evaluate performance of the test suite with both static and dynamic scheduling.
- We find that thread pool and OpenMP performed better than *Dieharder*. OpenMP performed similar or better than thread pool depending on the number of threads used.
- We conclude that for *Dieharder-T*, OpenMP with static scheduling offered the best performance.
- We propose a hybrid scheduling solution for *Dieharder-T* that utilizes the advantages of static and dynamic scheduling

1.1 Organization

The thesis is organized as follows. In Chapter 2, we provide the formal definition for PRNG and discuss a few existing statistical test suites. In Chapter 3, we described the design of the multi-threaded application and the platform used for the experiments. In Chapter 4, we discuss the experiments performed and an analysis of the results. In Chapter 5 we conclude and summarize future work.

CHAPTER 2:
Background

In this chapter, we review pseudorandom number generators (PRNGs) and provide a formal definition. We discuss a few existing statistical test suites before going into more details on two of interest to this study: *Diehard* and *Dieharder*.

2.1 Pseudorandom Number Generator

A PRNG is an implementation of an algorithm that generates a deterministic sequence of numbers that appears to be random. As these numbers are generated using an initial seed for the algorithm, the sequence is reproducible with the seed. This makes the sequence of numbers deterministic as long as the initial seed and algorithm used are known. If the sequence of numbers produced is interpreted as a sequence of bits, the PRNG may be called a deterministic random bit generator (DRBG). For the output of a generator to be deemed as *pseudorandom*, it "should be indistinguishable from a truly random sequence to an attacker" [7].

There are multiple applications for PRNGs, notably in simulation and cryptography. Bad PRNGs can cause misleading results in simulations [8]. The are many applications of PRNGs in cryptography, such as generating keys, initialization vectors and nonces. Low-entropy sequences in these applications often result in loss of security and attacks against implementations of cryptographic systems.

For a random sequence of numbers to be usable for cryptography, the sequence must be deemed uniformly distributed from the perspective of a computationally-bound adversary. Weakness in a PRNG can lead to cryptographic flaws, resulting in security risks. One example is the compromise of group chats of *Cryptocat* due to a programming flaw [1]. It was caused by a bug in the BigInt library which caused the random number generator to have a slight bias when generating keys, making it possible to crack these keys within a day [2]. Another example is a flaw with OpenSSL's entropy pool on Debian, reducing its strength from 256 bits to 15 bits [3]. It resulted in generating only 32,767 possible SSH keys of a given type and size, allowing brute force to be a practical attack on the key. More

recently, improper initialization of a PRNG led to android digital wallets being hijacked [4].

Adopting the definition of Desai, Hevia and Yin [7], we define a PRNG $\mathcal{GE}$ as a tuple of algorithms, $\mathcal{GE} = (\mathcal{K}, \mathcal{G})$. The *seed generating algorithm* $\mathcal{K}$ takes a security parameter k as input, to generate a *key* K and an *initial state* s_0. The *generation algorithm* $\mathcal{G}$ generates the *next state* s_i for $i \geq 1$ and an *output* y_i, using *key* K, the *current state* s_{i-1} and an *auxiliary input* t_i. The block length of the PRNG is the length of the PRNG output in each iteration, i.e., $n = |y_i|$ where y_i is a sequence of bits $y_i[0], y_i[1], \ldots, y_i[n-1]$.

A generator is considered *practical* if this sequence of bits is easy to generate. A generator is considered *secure* (or unpredictable) if this sequence of bits appears indistinguishable from random to any computationally-bound adversary. This notion can be formally described in several ways, but two common notions are the *next-bit-test* and *Yao's statistical test* [9]. In the former notion, no polynomial-time Turing machine has significant success in observing the first i bits of a sequence and accurately predicting bit $i+1$ in the sequence. In the latter notion, no statistical test represented by polynomial-sized circuit has significant success in differentiating the first i bits of the generated sequence from a i-bit long random sequence. Yao's theorem shows that these two notions are, in fact, related: a collection of i-bit sequences "passes the *next-bit-test* if and only if it passes all polynomial-sized *statistical tests*" [10].

It is often much easier to show at the design-level that a generator is provably secure in the *next-bit-test* sense. The notion that a generator is secure if no statistical test appears to exist differentiating it from random, however, is both intuitive and natural. As a result, statistical test suites have been developed which may be used to validate both design and implementation. Statistical test suites, however, provide a much weaker guarantee than the notion introduced by Yao, which covers all practical statistical tests (including those yet to have been imagined).

2.2 Statistical Test Suites

Many statistical tests have been proposed by different authors for testing PRNGs, and these have been collected into statistical test suites. For example, the National Institute for Standards and Technology (NIST) statistical test suite (STS) is a statistical test suite released in 2001, designed for testing cryptographically secure pseudorandom number gen-

erators (CSPRNGs). It consists of 16 simple statistical tests of non-randomness in binary sequences, intended as a baseline and reference implementation for statistical testing [11]. In 2007, L'Ecuyer and Simard released *TestU01* for statistical testing of PRNGs [12], providing some tests not previously implemented in other test suites. Next, we discuss two suites in more detail, as they are most relevant to this thesis: *Diehard* and *Dieharder*.

2.2.1 Diehard Test Suite

One of the most popular statistical test suites is Marsaglia's *Diehard* battery of tests, released in 1996. The suite includes 15 tests, written in C, translated from Fortran via the *f2c* utility. Detailed test descriptions are released with the test suite and not reproduced here [13], but the list of tests is reproduced in Table 2.1.

The tests require an input binary file of 32-bit integers to represent the sequence of random numbers [14]. For input files shorter than the required length for each test, the test will run until the end of the file, output an "END OF FILE" message and skip to the next test. All tests, except the *Runs* Test, require various parameters such as sample size, bit patterns, etc. These parameters have been preset to allow users to employ the test suite with ease and are not configurable by the user. If a user wishes to customize these parameters, they would have to modify the source code and recompile the test suite.

2.2.2 Dieharder Test Suite

Brown developed the *Dieharder* test suite as a GNU-licensed reimplementation of the *Diehard* test suite [15]. *Dieharder* tests are rewritten C code based on test descriptions from *Dieharder* and *NIST STS*. It also includes additional tests developed by Robert G. Brown and David Bauer. The test suite was named *Dieharder* both as a movie sequel pun as well as a tribute to George Marsaglia, author of the *Diehard* test suite.

Diehard and *Dieharder* are significantly different. The former uses only binary file for input to testing, requiring in the range of ten million random numbers [14]; the latter, however, prefers test generators written using the GNU scientific library (GSL) interface as to receive an unbounded stream of random numbers. Currently, 80 well-known generators are supported, many of which are drawn directly from the GSL. Support for reading raw input through a file is wrapped in a GSL-interface, also. The rationale for using this generator-

like, streaming API is to support larger sequence of random numbers. Modern applications may require much more than 10^{18} random numbers generated from millions of seeds; these may be sensitive to random number generator (RNG) weaknesses that might not be discovered by sequences limited to 10^7 random numbers. As of writing, the current version of *Dieharder* (3.31.1) has 31 fully implemented tests [16] (see Table 2.1), including tests from *Diehard*, *NIST STS*, tests designed by Brown and Bauer, as well as popular tests from other sources.

2.3 Test Suite Design

There are many parameters used in *Dieharder* that would reasonably be used in other tests suites as well. These parameters are used to control how many random values are tested in each individual test (psamples, tsamples, multiple_p), controlling the generators (seed, strategy) and control test (Xtrategy, Xoff, entity count, ksflag). Some of the parameters [17] used in *Dieharder* are explained here to understand how they affect the test results.

- psamples: Number of p-value samples per test.
- tsamples: Number of trials used in each test.
- multiply_p: Multiply the number of psamples for each test by a constant amount.
- seed: Initial seed value for PRNG.
- strategy: Reseeding strategy. The default (0) only reseeds at the start of program; non zero reseeds at start of each test.
- xtrategy: Strategy for when to stop test. The default (0), runs tests with a specified number of psamples and tsamples; resolve ambiguity mode reruns the test until ambiguity is resolved by adding psamples; test to destruction mode reruns the test until failure or reached max psamples.
- xoff: Max number of psamples to determine test is 'good'.
- ksflag : Which Kolmogorov-Smirnov test type to run. The default (0) is "fast but slightly sloppy" for psamples > 4999. A much slower but more accurate mode (1) is available for larger number of psamples, or (2) a very slow mode that is accurate to machine precision.
- ntuple: Set the ntuple length for tests on short bit strings that permit the length to be varied.

Table 2.1. Comparison of Tests Available in Diehard and Dieharder.

Diehard	**Dieharder**
Birthday Spacings	Diehard Birthday Test
Overlapping Permutations	Diehard OPERM5 Test
Ranks of 31x31 and 32x32 Matrices	Diehard 32x32 Binary Rank Test
Ranks of 6x8 Matrices	Diehard 6x8 Binary Rank Test
Monkey Tests on 20-bit Words	Diehard Bitstream Test
	Diehard OPSO Test
Monkey Tests OPSO, OQSO, DNA	Diehard OQSO Test
	Diehard DNA Test
Count the 1's in a Stream of Bytes	Diehard Count the 1's (stream) Test
Count the 1's in Specific Bytes	Diehard Count the 1's (byte) Test
Parking Lot Test	Diehard Parking Lot Test
Minimum Distance Test	Diehard Minimum Distance (2d Circle) Test
3D Spheres Test	Diehard 3d Sphere (Minimum Distance) Test
The Squeeze Test	Diehard Squeeze Test
Overlapping Sums Test	Diehard Sums Test
Runs Test	Diehard Runs Test
Craps Test	Diehard Craps Test
-	Marsaglia and Tsang GCD Test
-	STS Monobit Test
-	STS Runs Test
-	STS Serial Test (Generalized)
-	RGB Bit Distribution Test
-	RGB Generalized Minimum Distance Test
-	RGB Permutations Test
-	RGB Lagged Sum Test
-	RGB Kolmogorov-Smirnov Test
-	DAB Byte Distribution
-	DAB DCT
-	DAB Fill Tree Test
-	DAB Fill Tree 2 Test
-	DAB Monobit 2 Test

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Methodology

This chapter will discuss in detail on the methodology used for this thesis. The design of the multi-threaded application and platform used for the experiments is described.

3.1 Program Design

Dieharder version 3.31.1 was used as the code base to implement a modified multi-threaded *Dieharder* test suite named *Dieharder-T*. The main code modifications were the way individual statistical tests were scheduled to run, and modification to individual test logic to support parallel processing.

Each statistical test from *Dieharder* is contained within a single file that implements the base class *Test.c*. It contains the statistical test logic, input parameters as well as the test results (outputs). *Dieharder-T* separated the test logic and input parameters to *ITest.c*, and the outputs of the test to *OTest.c*.

Dieharder includes an "all-tests" mode to allow a convenient way to benchmark the PRNG being tested. This benchmark mode is more than just running all 31 tests in the test suite. It includes running various statistical tests by Brown with varying ntuple values to comprehensively stress test the generator. The list of tests and corresponding ntuple values used in this mode is shown in Table 3.1. A total of 80 statistical tests were ran when not running in resolve ambiguity (RA) or test to destruction (TTD) Xtrategy modes, which might increase the number of statistical tests to run depending on test results as explained in Section 2.3. To enable a fair comparison between *Dieharder* and *Dieharder-T*, *Dieharder-T* will execute the exact same 80 statistical tests when running in this benchmark mode.

To implement multi-threading in *Dieharder-T*, two different libraries for threading were evaluated. The first implementation used a POSIX thread pool library implemented by Mark Gondree [5], and the second implementation used OpenMP [6]. As the tests do not depend on results of other tests, they can be executed in parallel without altering the test results. In order to validate test results against *Dieharder*, the random numbers provided to

both *Dieharder* and *Dieharder-T* have to be the same to generate the same p-value. Since the sequence of tests executed in *Dieharder-T* would be different from *Dieharder* due to parallelization, the PRNG would be required to be reseeded at the beginning of each test to ensure that the same tsamples are provided to both *Dieharder* and *Dieharder-T*.

3.1.1 Dieharder-T

Using the thread pool library by Gondree, a job will be created for each statistical test to be executed and added to a common thread pool queue. Whenever a thread in the thread pool is available, the next job in the queue will be assigned to it. When all jobs in the queue have been completed, the experiment will terminate successfully. The experiment that uses the thread pool library will be referred to as *Dieharder-T* in the following chapters.

3.1.2 Dieharder-T-OMP

OpenMP provides compiler directives, library routines and environment variables to achieve parallelism in the program. The statistical tests will be executed in a *parallel for* loop directive, allowing OpenMP to schedule the tests to different threads. By default, OpenMP uses a static schedule that assigns loop iterations to threads for execution. In a 4-thread application with 80 loops, static scheduling will assign loops 0 to 19 to thread ID 0, loops 20 to 39 to thread ID 1, loops 40 to 59 to thread ID 2 and lastly loops 60 to 79 to thread ID 3. This scheduling policy favors tasks that have similar execution time, which might not be suitable for the test suite due to varying execution times for different statistical tests, which will be referred to as *Dieharder-T-OMP-S*. Another type of scheduling available in OpenMP is Dynamic scheduling. This scheduling mode will assign a loop iteration to an available thread, similar to how the thread pool library works. It allows a more balanced execution time across threads, but incur a higher processing overhead as it requires the thread to wait after each task to receive the next iteration to execute. The trade-off between higher overhead and balanced work load will be analyzed in the next chapter. Dynamic scheduling will be referred to as *Dieharder-T-OMP-D* in the following chapters.

3.2 Experiment Platform

The experiments were conducted on Amazon Web Services (AWS) Elastic Compute Cloud (EC2) [18] to provide a reproducible platform for anyone that is interested to validate the

experiment results. The deployment platform was a c3.xlarge [19] compute optimized instance with 4 virtual CPU cores. The instance used was not a dedicated instance and the physical hardware was shared among different users. As a result of hardware sharing, there were some high variance in experiment execution time between repetitions of the same statistical test. The affected experiments were repeated to verify experiment execution time.

The experiments were set up using a shell script to vary the parameters used, and repeat individual tests. The Linux *time* utility [20] was used to measure the time elapsed for each test, and the real (wall clock) time was used to measure execution time. This provided a fair comparison between a multi-threaded application and single thread application as it measured the absolute time taken to execute each experiment.

To generate the input file for testing of the binary file generator for *Dieharder*, we used the */dev/urandom* on the AWS EC2 to generate a 1 terabytes (TB) binary file on an Elastic Block Store (EBS) *sc1* 2 TB volume [21] to be attached to our EC2 instance. */dev/urandom* was used instead of */dev/random* due to the blocking nature of */dev/random* when the entropy pool is empty [22]. */dev/urandom* was used in the experiments as it was the recommended PRNG between the two PRNG except when used for long-lived keys [22], [23].

3.3 Reproducibility

In order to verify that the multi-threaded statistical test suite results were accurate, its results must be validated using *Dieharder* test results. To have *Dieharder* and *Dieharder-T* executing the exact tests, the initial seed used for PRNG must be explicitly declared so that they are not randomly generated.

As random numbers are assigned in a block at the start of the test, the order of the tests execution will affect the block of random numbers assigned in normal operation of the PRNG. Test options such as RA or TTD Xtrategy modes mentioned in Section 2.3 will also increase the number of random numbers needed for individual tests that cannot be determined until the test is completed. Due to the nature of multi-threaded application executing statistical tests in parallel, the order of assigning the block of random numbers will be different from that of *Dieharder*, which executes the statistical tests sequentially.

To have a fair comparison of execution time and to verify that *Dieharder-T* was implemented correctly, the experiments performed in this thesis were initialized with the same initial seed that were reseeded at the start of each statistical test.

Table 3.1. List of Tests and ntuple Values for "All-Tests" (Benchmark) Mode

Test	ntuple values
Diehard Birthday Test	0
Diehard OPERM5 Test	0
Diehard 32x32 Binary Rank Test	0
Diehard 6x8 Binary Rank Test	0
Diehard Bitstream Test	0
Diehard OPSO Test	0
Diehard OQSO Test	0
Diehard DNA Test	0
Diehard Count the 1's (stream) Test	0
Diehard Count the 1's Test (byte)	0
Diehard Parking Lot Test	0
Diehard Minimum Distance (2d Circle) Test	0
Diehard 3d Sphere (Minimum Distance) Test	0
Diehard Squeeze Test	0
Diehard Sums Test	0
Diehard Runs Test	0
Diehard Craps Test	0
Marsaglia and Tsang GCD Test	0
STS Monobit Test	0
STS Runs Test	0
STS Serial Test (Generalized)	0
RGB Bit Distribution Test	1 - 12
RGB Generalized Minimum Distance Test	2 - 5
RGB Permutations Test	2 - 5
RGB Lagged Sum Test	0 - 32
RGB Kolmogorov-Smirnov Test	0
Byte Distribution	0
DAB DCT	0
DAB Fill Tree Test	0
DAB Fill Tree 2 Test	0
DAB Monobit 2 Test	0

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Experiments and Analysis

This chapter will discuss the experiments performed in this thesis and an analysis of the results. The first three experiments were conducted to narrow down the parameters to be used for the other experiments. The next two experiments were conducted to compare the performance of *Dieharder* and *Dieharder-T*. The last two experiments were conducted to evaluate the effects of OpenMP scheduling policy. The full results can be found in the Appendix.

4.1 PRNG Source

This experiment was designed to compare the two different methods for passing random numbers to *Dieharder*. The experiment measured the execution time for running the "all-tests" benchmark mode using file-based random number input and unbounded random number stream. The two different PRNGs selected for the experiment were Mersenne Twister and raw file input as described in Section 3.2. To determine if the run-time taken for "all-tests" benchmark mode increases linearly when number of psamples were increased, the number of psamples were varied by changing the multiply_p value. The experiment also helped to select the PRNG for the other experiments. The parameters used for this experiment are listed in Table 4.1.

Table 4.1. List of Parameters for PRNG Comparison

Parameter	**Value**
Test Suites	*Dieharder*
Xtrategy Mode	Normal (0)
KS Test	Default (0)
Tests	All-tests (a)
Generators	Mersenne Twister (13), Raw File Input (201)
Multiply P	1, 2, 4, 8

The run-time for Mersenne Twister and raw file input generators are shown in Figure 4.1 and the full results are shown in Appendix A.1. The results show a consistent rate of increase for the run-time when multiple_p is increased: the run-time doubled when psamples

was doubled for the Mersenne Twister. The total run-time of raw file input generator was much higher compared to the Mersenne Twister generator. The Mersenne Twister generator was selected as the generator to be used for the rest of the experiments as it would reduce total run-time by a factor of 7.

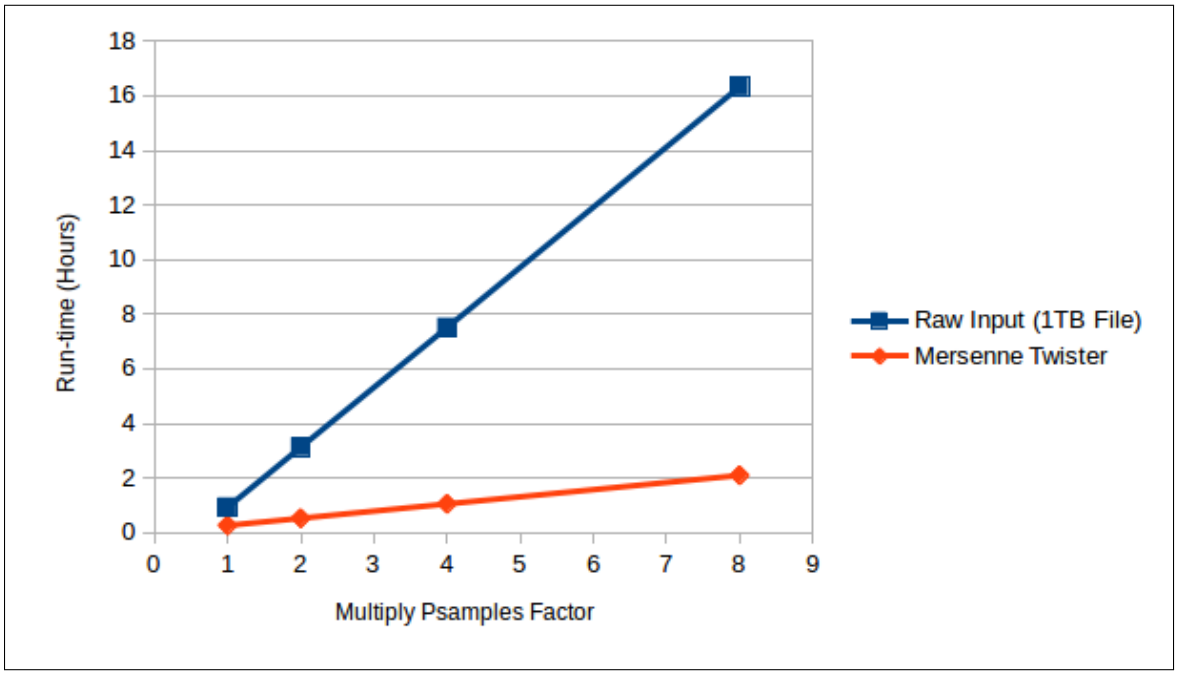


Figure 4.1. *Dieharder* Run-time for Different GSL Generators

4.2 Test to Destruction Xtrategy Mode

This purpose of this experiment was to estimate the time taken to execute TTD Xtrategy mode when multiply_p was increased. This experiment was the only experiment in this thesis that did not use the "all-tests" benchmark mode, as testing under TTD mode approached more than a week to complete with "all-tests". This experiment ran "*Diehard* bitstream test" with the parameters shown in Table 4.2. As RA and TTD modes required the Kolmogorov-Smirnov test (KSTEST) flag to use the accurate mode (2) [17], all three Xtrategy modes were executed in KSTEST accurate mode for consistency.

Table 4.2. List of Parameters for Test To Destruction Run-time Estimation

Parameter	**Value**
Test Suites	*Dieharder*
Xtrategy Mode	Default (0), Resolve Ambiguity (1), Test to Destruction (2)
KS Test	Accurate (2)
Tests	Diehard Bitstream Test (4)
Generators	Mersenne Twister (13)
Multiply P	1, 2, 4, 8

From the results as shown in Figure 4.2, it can be concluded that the run-time for TTD was nearly constant, even when multiply_p was increased. This is not surprising as individual statistical tests were repeated with psamples in increments of 100 until the PRNG failed the test or reached the maximum psamples as defined by the xoff parameter. As the run-time for individual statistical tests increased with increasing psample values, the run-time for smaller psample values only contributed a small percentage of total run-time for the experiment. Statistical tests with multiply_p value of 2 (starting psample of 200) performed only one less repetition (at default, psample value of 100, xoff value of 100,000, increment of 100) resulting in insignificant reduction in total run-time as multipy_p was increased.

The results also showed that the run-time for default and RA Xtrategy modes was much shorter compared to that for TTD mode. As shown in Appendix A.2, the run-time for TTD was more than 300 times longer when compared to default mode when multiply_p was 8. Using the results from the previous experiment, the estimated run-time for TTD in "all-tests" benchmark mode is at least 100 days. Since the time taken to run TTD would be excessive, no further experiments were executed in this Xtrategy mode. The other two modes were then evaluated using the "all-tests" benchmark mode in the next experiment as they showed similar execution time for just one statistical test.

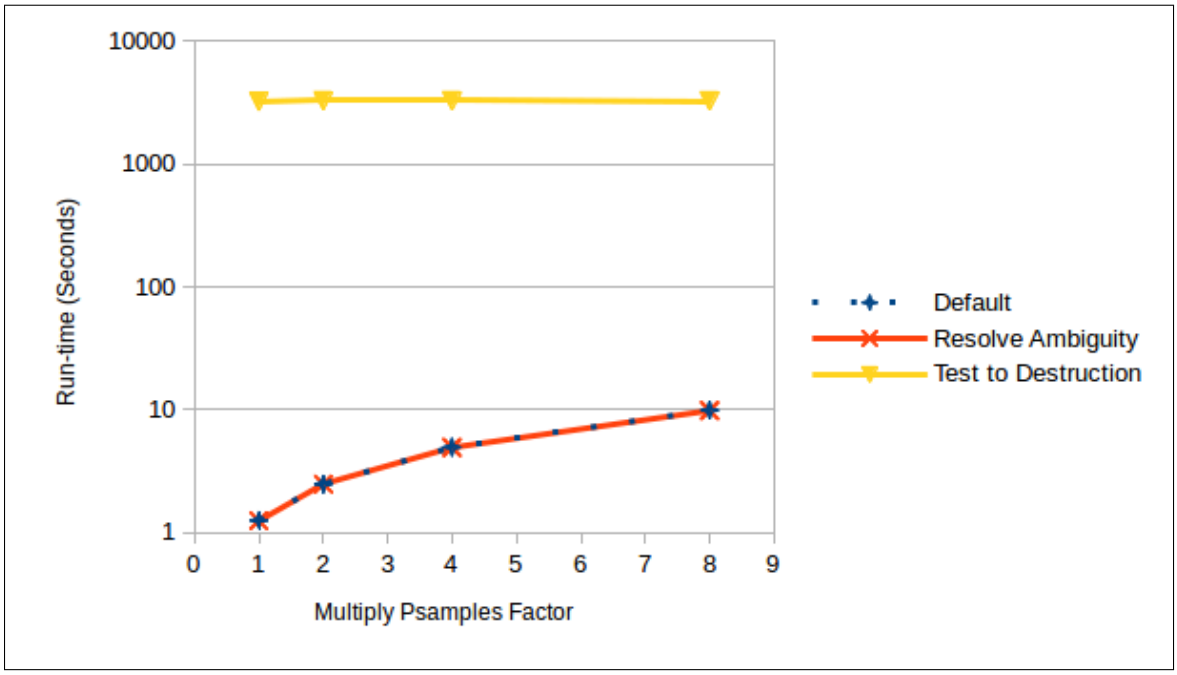


Figure 4.2. *Dieharder* Different Xtrategy Mode Run-time for Bitstream Test.
Y-Axis in logarithmic time scale.

4.3 Xtrategy Mode Selection

This goal of this experiment was to evaluate the default and RA Xtrategy modes to determine the Xtrategy mode to be used for the rest of the experiments. The experiment was executed using the "all-tests" benchmark mode with the parameters listed in Table 4.3.

Table 4.3. List of Parameters for Xtrategy Mode Selection

Parameter	Value
Test Suites	*Dieharder*
Xtrategy Mode	Default (0), Resolve Ambiguity (1)
KS Test	Accurate (2)
Tests	All-tests (a)
Generators	Mersenne Twister (13)
Multiply P	1, 2, 4, 8

As shown in Figure 4.3, both Xtrategy modes had similar execution time except when multiple_p had a value of 4. At multiply_p value 4, 32 individual statistical tests were

repeated in RA mode to resolve ambiguity, resulting in the additional 9 minutes compared to default mode as shown in Appendix A.3.

As run-time for the default Xtrategy mode was nearly identical but less variable than RA mode, the default was selected as the Xtrategy mode for the rest of the experiments. The run-time for RA is expected to follow closely to that of default mode when there are few ambiguous (weak) results.

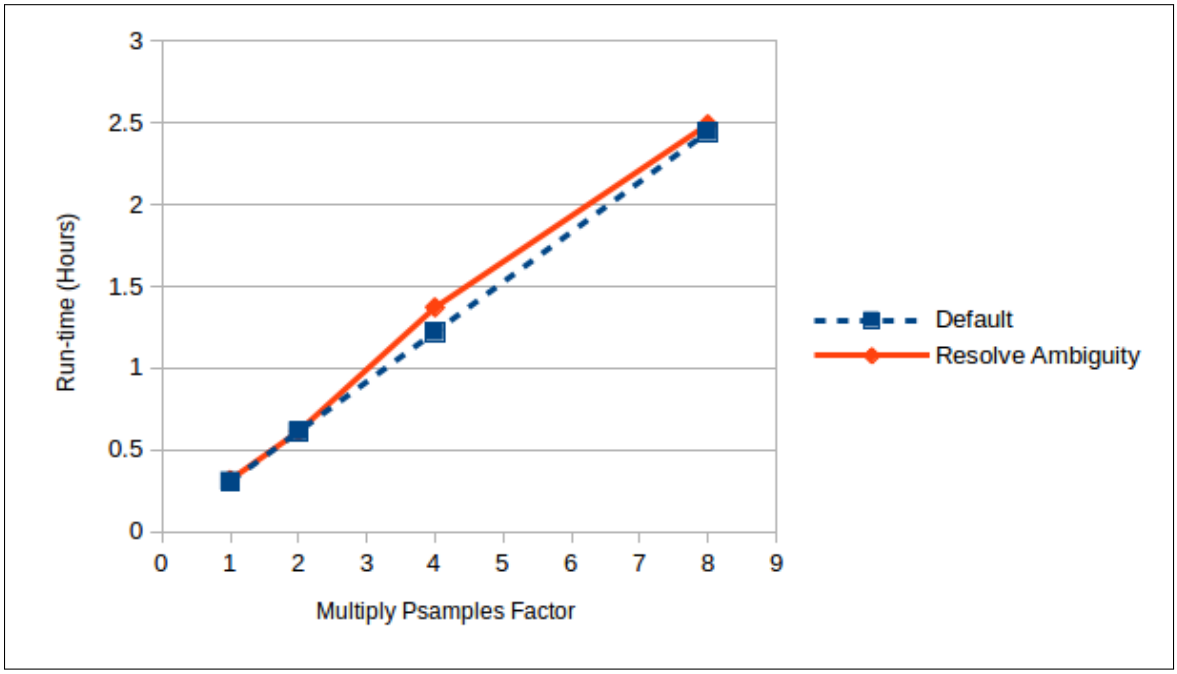


Figure 4.3. "All-Tests" Run-time for Default and Resolve Ambiguity Modes

4.4 *Dieharder* and *Dieharder-T* Comparison

This purpose of this experiment was to compare the run-time of *Dieharder* with the following variants of *Dieharder-T*:

- *Dieharder-T* in serial mode.
- Thread pool *Dieharder-T* with 1 to 4 thread counts.
- Static scheduling OpenMP *Dieharder-T* with 1 to 4 thread counts.

The parameters used for this experiment are shown in Table 4.4.

Table 4.4. List of Parameters for Statistical Test Suites Comparison

Parameter	Value
Test Suites	*Dieharder*, *Dieharder-T*, *Dieharder-T-OMP-S*
Xtrategy Mode	Default (0)
KS Test	Default (0)
Tests	All-tests (a)
Generators	Mersenne Twister (13)
Multiply P	1, 2, 4, 8

As shown in Figure 4.4, *Dieharder* had the longest run-time followed closely by *Dieharder-T* running in serial mode. This showed that the modifications made to *Dieharder* described in Section 3.1 improved code efficiency, reducing run-time as multiply_p was increased.

The run-time for 1-thread *Dieharder-T* and *Dieharder-T-OMP-S* were almost similar to serial *Dieharder-T* as expected.

The run-time for 2-thread *Dieharder-T* was significantly higher than 2-thread *Dieharder-T-OMP-S*, probably due to the dynamic scheduling overhead in thread pool. The run-time for 3-thread and 4-thread *Dieharder-T* was similar to that for 2-thread and 3-thread *Dieharder-T-OMP-S*. As thread pool uses a thread for polling of job statuses, the maximum number of central processing unit (CPU) cores available for executing statistical tests was limited to 3 in the experiment platform with 4 virtual CPUs. This resulted in 3-thread and 4-thread *Dieharder-T* having similar run-time.

The 4-thread *Dieharder-T-OMP-S* had the shortest run-time for all statistical test suites evaluated and was 16% faster than 3-thread *Dieharder-T-OMP-S*. The 3-thread *Dieharder-T-OMP-S* was only about 2% faster than 2-thread *Dieharder-T-OMP-S* and is examined in greater details in Section 4.5. The 2-thread *Dieharder-T-OMP-S* took almost half the time of 1-thread *Dieharder-T-OMP-S* as expected since the tasks were split between two threads.

The results showed that *Dieharder-T-OMP-S* performed the same or better than *Dieharder-T* depending on the number of threads used, making it the preferred choice for implementing multi-threading for the PRNG test suite.

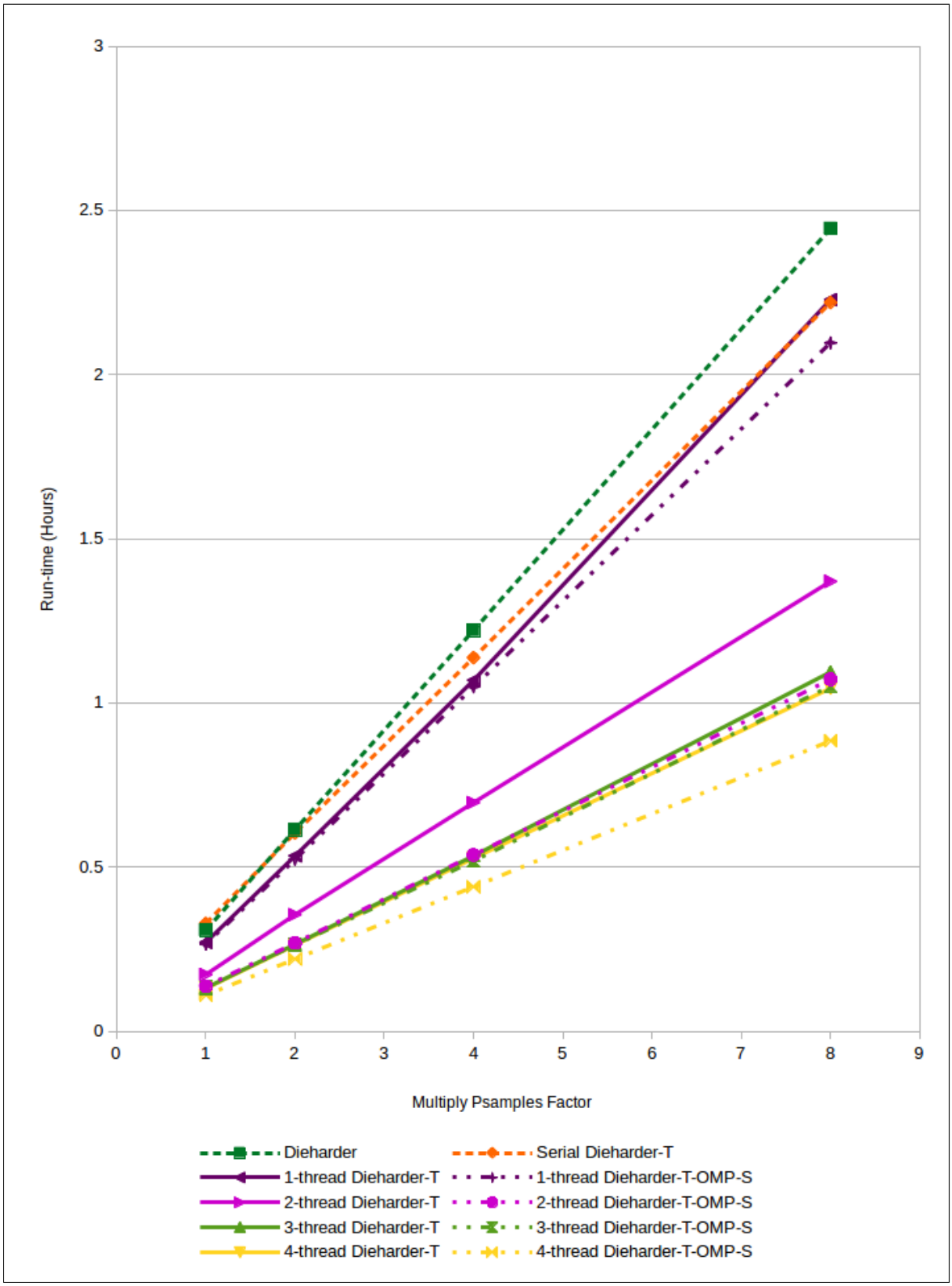


Figure 4.4. "All-Tests" Run-time for Different Statistical Test Suites

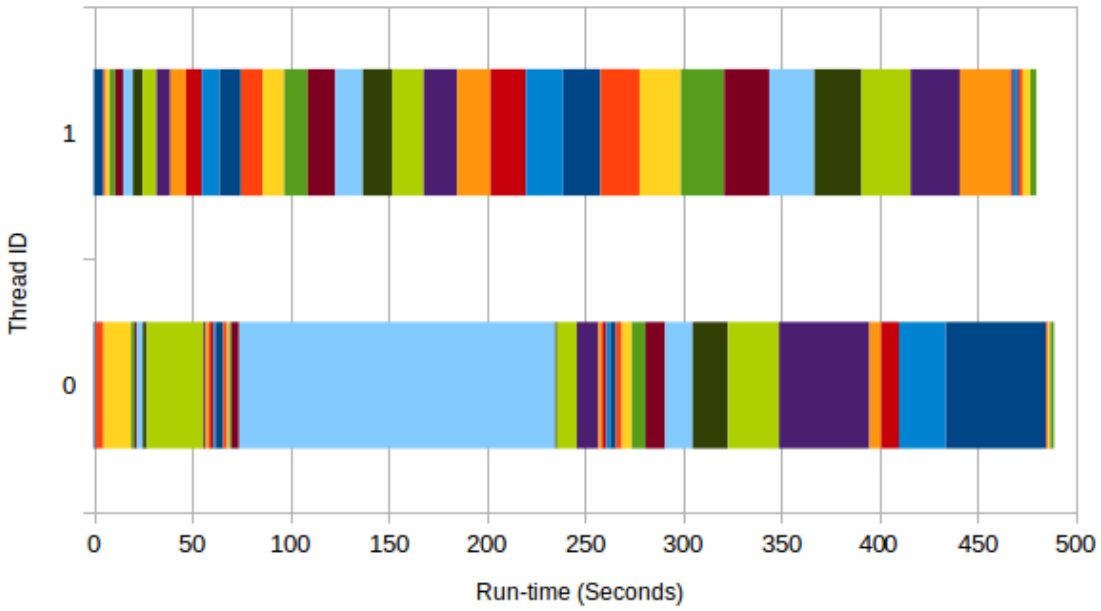
4.5 Static Scheduling Individual Test Run-time

The goal of this experiment was to collect the run-time for individual statistical tests in each thread for 2-thread and 3-thread *Dieharder-T-OMP-S*. It was to determine if the similarity in run-time despite having a difference of one thread was due to inefficient allocation of statistical tests to threads. The parameters used to run this experiment can be found in Table 4.5

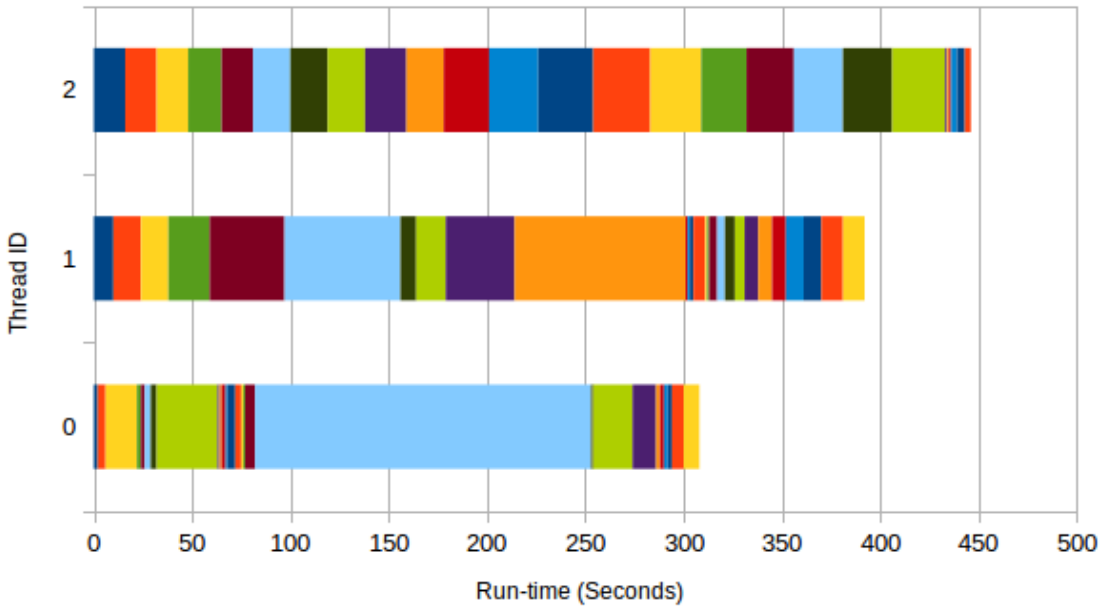
Table 4.5. List of Parameters for Static Scheduling Run-time Collection

Parameter	**Value**
Test Suites	*Dieharder-T-OMP-S*
Number of Threads	2, 3
Xtrategy Mode	Default (0)
KS Test	Default (0)
Tests	All-tests (a)
Generators	Mersenne Twister (13)
Multiply P	1

The run-time for individual statistical tests in 2-thread and 3-thread *Dieharder-T-OMP-S* are shown in Figure 4.5. The colors are used to denote the order of statistical tests within a thread. The same color does not represent the same test in 2-thread and 3-thread *Dieharder-T-OMP-S*. Figure 4.5a shows that the run-time between both threads are almost equal and Figure 4.5b shows that the run-time between all 3 threads varied greatly. Since the run-time of the test suite is determined by the thread with the longest run-time, the unequal distribution of run-time load among the threads led to only 2% reduction in total run-time despite adding an additional thread. The experiment described in Section 4.6 was conducted to determine if changing OpenMP scheduling policy would improve the overall performance regardless of thread counts.



(a) 2-Thread Static Scheduling



(b) 3-Thread Static Scheduling

Figure 4.5. Individual Statistical Tests Run-time for Individual Thread

The colors are used to denote order of statistical tests run within a thread. They do not represent the same tests within (a) and (b).

4.6 OpenMP Scheduling Policy

In this experiment, different OpenMP scheduling policies explained in Section 3.1.2 were evaluated to determine their impact on statistical test suite run-time. Static and dynamic scheduling policies were executed using the parameters shown in Table 4.6.

Table 4.6. List of Parameters for Scheduling Policy Comparison

Parameter	Value
Test Suites	*Dieharder-T-OMP-S*, *Dieharder-T-OMP-D*
Xtrategy Mode	Default (0)
KS Test	Default (0)
Tests	All-tests (a)
Generators	Mersenne Twister (13)
Multiply P	1, 2, 4, 8

The run-time for both scheduling policies are shown in Figure 4.6. The results show that 1-thread *Dieharder-OMP-S* and 1-thread *Dieharder-OMP-D* had similar run-time. This was expected as all tasks could only be allocated to 1 thread, there should be no difference between static and dynamic scheduling.

2-thread *Dieharder-OMP-S* had shorter run-time compared to 2-thread *Dieharder-OMP-D*. This could be attributed to dynamic scheduling having a higher overhead and 2-thread static scheduling being more efficient in this instance as shown in Figure 4.5a.

When the thread count was increased to 3, *Dieharder-OMP-D* performed slightly better than *Dieharder-OMP-S*. As seen from Figure 4.5b, 3-thread static scheduling was inefficient when allocating tasks to threads. The overhead incurred in dynamic scheduling was less costly compared to the inefficient allocation of tasks.

When the thread count was increased to 4, *Dieharder-OMP-D* did not increase in performance compared to *Dieharder-OMP-S*, with both achieving similar run-time. The experiment described in Section 4.5 was conducted to determine why both scheduling polices had similar performance.

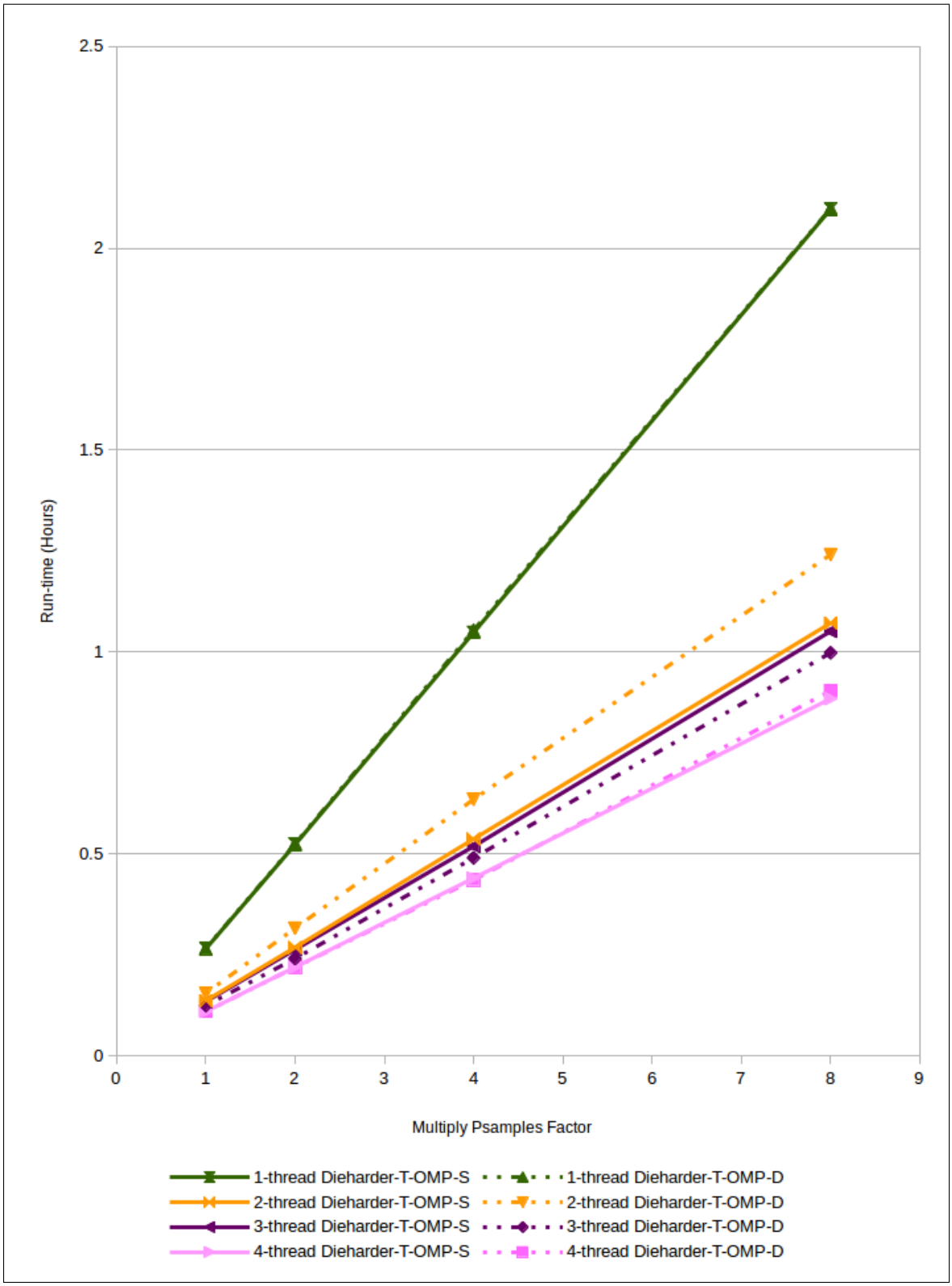


Figure 4.6. OMP Dynamic Scheduling Run-time

4.7 4-thread Scheduling Individual Run-time

This experiment was performed to collect the run-time for individual statistical tests in each thread for 4-thread static and dynamic scheduling. This was to identify why the run-time for 4-thread static and dynamic scheduling were similar as described in Section 4.6. The parameters used to run this experiment are shown in Table 4.7.

Table 4.7. List of Parameters for 4-thread Individual Run-time Collection

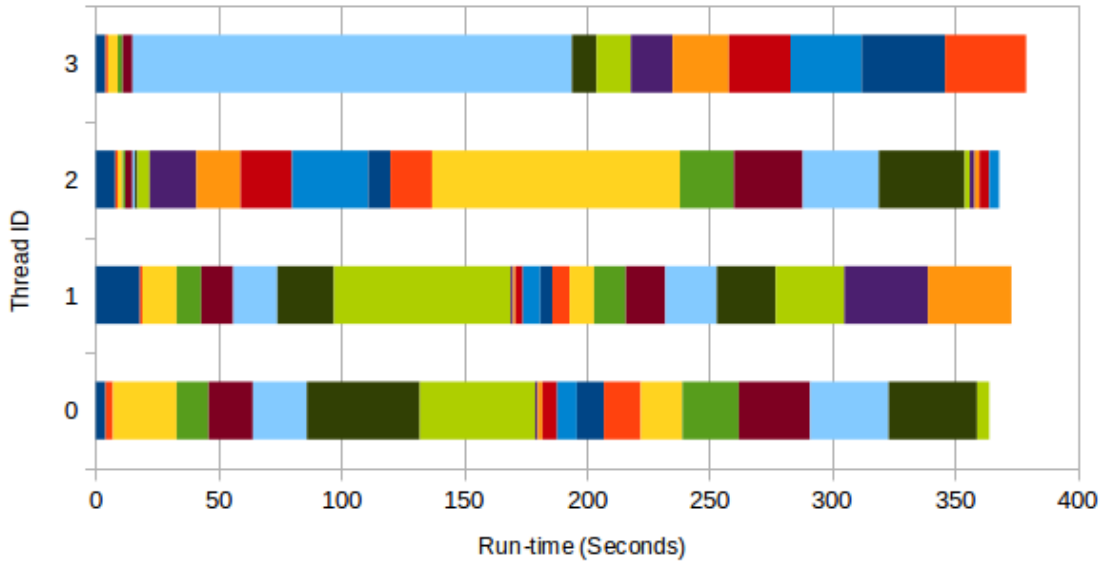
Parameter	Value
Test Suites	*Dieharder-T-OMP-S*, *Dieharder-T-OMP-DS*
Number of Threads	4
Xtrategy Mode	Default (0)
KS Test	Default (0)
Tests	All-tests (a)
Generators	Mersenne Twister (13)
Multiply P	1, 2, 4, 8

As shown in Figure 4.7, dynamic scheduling allocated statistical tests to threads very efficiently. Dynamic scheduling is good when tasks did not have consistent run-time, similar to individual statistical tests having different experimental run-times. The disadvantage of dynamic scheduling is that it incurred higher overhead as it had to pause the thread every time it had to allocate a new task. The purpose of this experiment was determining if the extra overhead incurred was less than the time saved from efficient allocation of tasks.

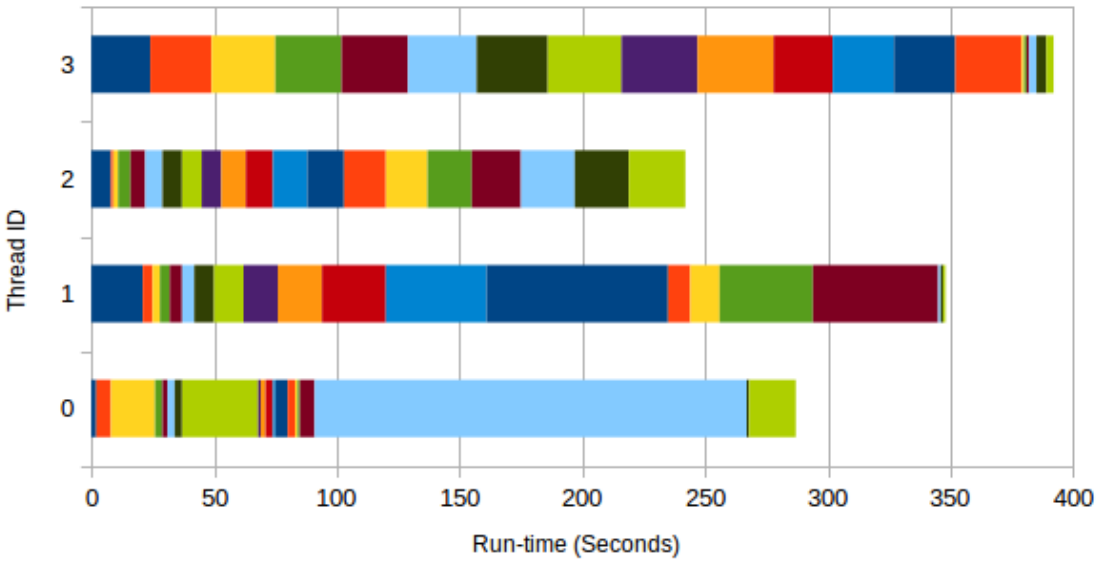
The run-times for static scheduling are shown in Figure 4.7b. Although the same number of tasks were assigned to each thread, the time taken to complete all tasks varied widely among all four threads. The faster threads were idle for 43%, 30% and 13% for the duration of the experiment, showing clear inefficiency in task allocation.

Using both results, it can be concluded that dynamic scheduling was able to allocate tasks efficiently among the thread, and static scheduling was able to allocate tasks to threads with little overhead. It was concluded that the most efficient way to implement multi-threading would be to apply static scheduling to tasks that are arranged such that tasks add up to similar run-time. This approach would require each statistical test to be benchmarked and assigned a value indicating time required. This would allow the program to calculate the estimated run-time for each thread and arrange the tasks such that each thread would get

tasks adding up to the estimated run-time.



(a) Dynamic Scheduling



(b) Static Scheduling

Figure 4.7. Run-time for Different *Dieharder-T* OpenMP Scheduling Types

The colors are used to denote order of statistical tests run within a thread. They do not represent the same tests within (a) and (b).

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusion

The primary contributions of this thesis are as follows:

- We modified *Dieharder* to support multi-threading, creating a variant test-suite (*Dieharder-T*).
- We integrated Gondree's thread pool library into *Dieharder-T* and evaluated performance of the test suite.
- We integrated OpenMP into *Dieharder-T* and evaluate performance of the test suite with both static and dynamic scheduling.

Our experimental results show that multi-threading reduced the run-time of *Dieharder*. Due to inefficient scheduling in *Dieharder*, multi-threading showed a decrease in run-time when the number of threads were increased. OpenMP with static scheduling showed the most consistent reduction of run-time when compared to *Dieharder*.

The run-time for static scheduling was halved when the number of threads were increased to two. This was achieved as the run-time between both threads were similar, resulting in halving of total run-time. The run-time was not halved further when the numbers of threads were increased to four as there was great disparity among thread run-time.

The results showed that dynamic scheduling did not provide better performance to static scheduling due to the overheads incurred when allocating tasks to threads. 2-thread dynamic scheduling did not halve the run-time of 1-thread dynamic scheduling due to the overhead incurred. The overhead appeared consistent even as number of threads increased, resulting in declining performance when number of threads were increased. This made dynamic scheduling unsuitable to be used for multi-threading in our PRNG test suite.

Based on the conclusions described in Section 4.7, efficient allocation of tasks can be achieved by combining the advantage from both types of scheduling policies. This would allow the reduction in run-time to be more consistent when the number of threads are increased. By rearranging the tasks such that tasks in each thread add up to similar run-times, the threads would not idle for long. This approach could use the data from Appendix A.6,

assigning an estimated run-time to each statistical test. An additional sorting stage would be required to be added before tasks are allocated to threads with static scheduling. The recommended change would require an estimation of run-time to be assigned to each statistical test and a sorting stage to arrange the statistical tests for assignment to each thread. This would assign tasks to threads using static scheduling and achieving low variance of run-time between threads similar to dynamic scheduling.

APPENDIX: Experiments Result

This appendix contains the raw experimental data that was collected during the experiments described in Chapter 4. The "change factor" shown in the tables below is the change in run-time when the number of psample values are doubled.

A.1 *Dieharder* Generators Run-time

This section contains the raw experimental data that was collected for *Dieharder* generators.

Table A.1. Run-time for Mersenne Twister Generator

Multiply_P	**Run 1 (S)**	**Run 2 (S)**	**Run 3 (S)**	**Average (Hours)**	**Change Factor**
1	946	941	947	0.26	-
2	1887	1887	1887	0.52	2.00
4	3775	3778	3776	1.05	2.00
8	7549	7544	7540	2.10	2.00

Table A.2. Run-time for Raw File Input Generator

Multiply_P	**Run 1 (S)**	**Run 2 (S)**	**Run 3 (S)**	**Average (Hours)**	**Change Factor**
1	2590	3789	3647	0.93	-
2	6815	13607	13343	3.13	3.37
4	15340	32972	32822	7.51	2.40
8	33456	71390	71525	16.33	2.17

A.2 Bitstream Test Run-time

This section contains the raw experimental data that was collected for Bitstream test.

Table A.3. Bitstream Test Run-time for Xtrategy Modes

Multiply_P	**Run 1 (S)**	**Run 2 (S)**	**Run 3 (S)**	**Average (S)**	**Change Factor**
Default					
1	1.28	1.25	1.25	1.26	-
2	2.48	2.48	2.48	2.48	1.97
4	4.96	4.95	4.95	4.95	2.00
8	9.91	9.91	9.90	9.91	2.00
Resolve Ambiguity					
1	1.24	1.25	1.25	1.25	-
2	2.49	2.49	2.48	2.49	2.00
4	4.96	4.96	4.87	4.93	1.98
8	9.76	9.90	9.82	9.82	1.99
Test to Destruction					
1	3247	3139	3271	3235	-
2	3258	3287	3302	3282	1.01
4	3268	3267	3276	3277	1.00
8	3259	3274	3278	3273	1.00

A.3 Xtrategy Modes Run-time

This section contains the raw experimental data that was collected for different Xtrategy modes.

Table A.4. "All-Tests" Run-time for Xtrategy Modes

Multiply_P	**Run 1 (S)**	**Run 2 (S)**	**Run 3 (S)**	**Average (Hours)**	**Change Factor**
Default					
1	1106	1110	1106	0.31	-
2	2216	2206	2207	0.61	2.00
4	4390	4383	4417	1.22	1.99
8	8792	8752	8864	2.45	2.00
Resolve Ambiguity					
1	1143	1143	1150	0.32	-
2	2209	2214	2209	0.61	1.93
4	4933	4959	4923	1.37	2.23
8	8990	8958	8956	2.49	1.82

A.4 Statistical Test Suites Run-time

This section contains the raw experimental data that was collected for different statistical test suites.

Table A.5. Statistical Test Suite Run-time - Non-threaded

Multiply_P	**Run 1 (S)**	**Run 2 (S)**	**Run 3 (S)**	**Average (Hours)**	**Change Factor**
Dieharder					
1	1106	1110	1106	0.31	-
2	2216	2206	2207	0.61	2.00
4	4390	4383	4417	1.22	1.99
8	8792	8752	8864	2.45	2.00
Dieharder-T Serial					
1	923	1307	1313	0.33	-
2	2488	2108	1928	0.60	1.84
4	3828	4404	4055	1.14	1.88
8	7449	8557	7964	2.22	1.95

Table A.6. Statistical Test Suite Run-time - Dieharder-T

Multiply_P	**Run 1 (S)**	**Run 2 (S)**	**Run 3 (S)**	**Average (Hours)**	**Change Factor**
Dieharder-T 1-Thread					
1	975	973	973	0.27	-
2	1919	1922	1927	0.53	1.97
4	3847	3829	3849	1.07	2.00
8	7672	8714	7677	2.23	2.08
Dieharder-T 2-Thread					
1	618	617	627	0.17	-
2	1348	1252	1236	0.36	2.06
4	2485	2466	2570	0.70	1.96
8	5031	4731	5038	1.37	1.97
Dieharder-T 3-Thread					
1	476	472	474	0.13	-
2	954	952	951	0.26	2.01
4	1882	1881	2006	0.53	2.02
8	3950	4058	3807	1.09	2.05
Dieharder-T 4-Thread					
1	463	476	466	0.13	-
2	945	950	939	0.26	2.02
4	1903	1893	1894	0.53	2.01
8	3727	3812	3743	1.04	1.99

Table A.7. Statistical Test Suite Run-time - Dieharder-T-OMP-S

Multiply_P	**Run 1 (S)**	**Run 2 (S)**	**Run 3 (S)**	**Average (Hours)**	**Change Factor**
Dieharder-T-OMP-S 1-Thread					
1	955	956	954	0.27	-
2	1884	1888	1881	0.52	1.97
4	3774	3777	3773	1.05	2.00
8	7572	7534	7541	2.10	2.00
Dieharder-T-OMP-S 2-Thread					
1	494	492	492	0.14	-
2	970	966	965	0.27	1.96
4	1934	1935	1931	0.54	2.00
8	3857	3860	3859	1.07	2.00
Dieharder-T-OMP-S 3-Thread					
1	500	493	462	0.13	-
2	962	957	924	0.26	1.95
4	1811	1984	1809	0.52	1.97
8	3655	3991	3710	1.05	2.03
Dieharder-T-OMP-S 4-Thread					
1	392	401	394	0.11	-
2	791	792	791	0.22	2.00
4	1567	1570	1617	0.44	2.00
8	3207	3145	3201	0.88	2.01

Table A.8. Statistical Test Suite Run-time - Dieharder-T-OMP-D

Multiply_P	Run 1 (S)	Run 2 (S)	Run 3 (S)	Average (Hours)	Change Factor
Dieharder-T-OMP-D 1-Thread					
1	964	958	952	0.27	-
2	1881	1882	1909	0.53	1.97
4	3801	3790	3790	1.05	2.01
8	7543	7557	7562	2.10	2.00
Dieharder-T-OMP-D 2-Thread					
1	565	557	552	0.16	-
2	1102	1163	1150	0.32	2.04
4	2259	2301	2300	0.64	2.01
8	4384	4434	4591	1.24	1.95
Dieharder-T-OMP-D 3-Thread					
1	438	453	455	0.12	-
2	855	885	863	0.24	1.93
4	1709	1754	1832	0.49	2.03
8	3428	3618	3734	1.00	2.04
Dieharder-T-OMP-D 4-Thread					
1	395	397	394	0.11	-
2	788	792	790	0.22	2.00
4	1630	1479	1598	0.44	1.99
8	3190	3294	3282	0.90	2.07

A.5 Static Scheduling Run-time Per Thread

This section contains the raw experimental data that was collected for 2-thread and 3-thread static scheduling.

Table A.9. 2-Thread Static Scheduling Tests Run-time - Thread 0

Thread ID	Statistical Test	ntuple	Run-time (Seconds)
0	Diehard Birthday Test	0	1
0	Diehard OPERM5 Test	0	4
0	Diehard 32x32 Binary Rank Test	0	14
0	Diehard 6x8 Binary Rank Test	0	2
0	Diehard Bitstream Test	0	1
0	Diehard OPSO Test	0	3
0	Diehard OQSO Test	0	2
0	Diehard DNA Test	0	29
0	Diehard Count the 1's (stream) Test	0	1
0	Diehard Count the 1's (byte) Test	0	2
0	Diehard Parking Lot Test	0	2
0	Diehard Minimum Distance (2d Circle) Test	2	1
0	Diehard 3d Sphere (Minimum Distance) Test	3	4
0	Diehard Squeeze Test	0	2
0	Diehard Sums Test	0	1
0	Diehard Runs Test	0	1
0	Diehard Craps Test	0	4
0	Marsaglia and Tsang GCD Test	0	161
0	STS Monobit Test	1	1
0	STS Runs Test	2	10
0	STS Serial Test	1	11
0	RGB Bit Distribution Test	1	2
0	RGB Bit Distribution Test	2	2
0	RGB Bit Distribution Test	3	2
0	RGB Bit Distribution Test	4	3
0	RGB Bit Distribution Test	5	3
0	RGB Bit Distribution Test	6	5
0	RGB Bit Distribution Test	7	7
0	RGB Bit Distribution Test	8	10
0	RGB Bit Distribution Test	9	14
0	RGB Bit Distribution Test	10	18
0	RGB Bit Distribution Test	11	26
0	RGB Bit Distribution Test	12	46
0	RGB Generalized Minimum Distance Test	2	6
0	RGB Generalized Minimum Distance Test	3	9
0	RGB Generalized Minimum Distance Test	4	24
0	RGB Generalized Minimum Distance Test	5	51
0	RGB Permutations Test	2	1
0	RGB Permutations Test	3	1
0	RGB Permutations Test	4	2
	Total		489

Table A.10. 2-Thread Static Scheduling Tests Run-time - Thread 1

Thread ID	Statistical Test	ntuple	Run-time (Seconds)
1	RGB Permutations Test	5	5
1	RGB Lagged Sum Test	0	1
1	RGB Lagged Sum Test	1	2
1	RGB Lagged Sum Test	2	3
1	RGB Lagged Sum Test	3	4
1	RGB Lagged Sum Test	4	5
1	RGB Lagged Sum Test	5	5
1	RGB Lagged Sum Test	6	7
1	RGB Lagged Sum Test	7	7
1	RGB Lagged Sum Test	8	8
1	RGB Lagged Sum Test	9	8
1	RGB Lagged Sum Test	10	9
1	RGB Lagged Sum Test	11	11
1	RGB Lagged Sum Test	12	11
1	RGB Lagged Sum Test	13	11
1	RGB Lagged Sum Test	14	12
1	RGB Lagged Sum Test	15	14
1	RGB Lagged Sum Test	16	14
1	RGB Lagged Sum Test	17	15
1	RGB Lagged Sum Test	18	16
1	RGB Lagged Sum Test	19	17
1	RGB Lagged Sum Test	20	17
1	RGB Lagged Sum Test	21	18
1	RGB Lagged Sum Test	22	19
1	RGB Lagged Sum Test	23	19
1	RGB Lagged Sum Test	24	20
1	RGB Lagged Sum Test	25	21
1	RGB Lagged Sum Test	26	22
1	RGB Lagged Sum Test	27	23
1	RGB Lagged Sum Test	28	23
1	RGB Lagged Sum Test	29	24
1	RGB Lagged Sum Test	30	25
1	RGB Lagged Sum Test	31	25
1	RGB Lagged Sum Test	32	26
1	RGB Kolmogorov-Smirnov Test	0	1
1	DAB Byte Distribution	0	2
1	DAB DCT	256	1
1	DAB Fill Tree Test	32	2
1	DAB Fill Tree 2 Test	0	4
1	DAB Monobit 2 Test	12	3
	Total		480

Table A.11. 3-Thread Static Scheduling Tests Run-time - Thread 0

Thread ID	Statistical Test	ntuple	Run-time (Seconds)
0	Diehard Birthday Test	0	2
0	Diehard OPERM5 Test	0	4
0	Diehard 32x32 Binary Rank Test	0	16
0	Diehard 6x8 Binary Rank Test	0	2
0	Diehard Bitstream Test	0	2
0	Diehard OPSO Test	0	3
0	Diehard OQSO Test	0	3
0	Diehard DNA Test	0	31
0	Diehard Count the 1's (stream) Test	0	1
0	Diehard Count the 1's (byte) Test	0	1
0	Diehard Parking Lot Test	0	2
0	Diehard Minimum Distance (2d Circle) Test	2	1
0	Diehard 3d Sphere (Minimum Distance) Test	3	4
0	Diehard Squeeze Test	0	3
0	Diehard Sums Test	0	1
0	Diehard Runs Test	0	1
0	Diehard Craps Test	0	5
0	Marsaglia and Tsang GCD Test	0	171
0	STS Monobit Test	1	1
0	STS Runs Test	2	20
0	STS Serial Test	1	12
0	RGB Bit Distribution Test	1	2
0	RGB Bit Distribution Test	2	2
0	RGB Bit Distribution Test	3	2
0	RGB Bit Distribution Test	4	2
0	RGB Bit Distribution Test	5	6
0	RGB Bit Distribution Test	6	8
	Total		308

Table A.12. 3-Thread Static Scheduling Tests Run-time - Thread 1

Thread ID	Statistical Test	ntuple	Run-time (Seconds)
1	RGB Bit Distribution Test	7	10
1	RGB Bit Distribution Test	8	14
1	RGB Bit Distribution Test	9	14
1	RGB Bit Distribution Test	10	21
1	RGB Bit Distribution Test	11	38
1	RGB Bit Distribution Test	12	59
1	RGB Generalized Minimum Distance Test	2	8
1	RGB Generalized Minimum Distance Test	3	15
1	RGB Generalized Minimum Distance Test	4	35
1	RGB Generalized Minimum Distance Test	5	87
1	RGB Permutations Test	2	1
1	RGB Permutations Test	3	1
1	RGB Permutations Test	4	2
1	RGB Permutations Test	5	6
1	RGB Lagged Sum Test	0	1
1	RGB Lagged Sum Test	1	1
1	RGB Lagged Sum Test	2	4
1	RGB Lagged Sum Test	3	4
1	RGB Lagged Sum Test	4	5
1	RGB Lagged Sum Test	5	5
1	RGB Lagged Sum Test	6	7
1	RGB Lagged Sum Test	7	7
1	RGB Lagged Sum Test	8	7
1	RGB Lagged Sum Test	9	9
1	RGB Lagged Sum Test	10	9
1	RGB Lagged Sum Test	11	11
1	RGB Lagged Sum Test	12	11
	Total		392

Table A.13. 3-Thread Static Scheduling Tests Run-time - Thread 2

Thread ID	Statistical Test	ntuple	Run-time (Seconds)
2	RGB Lagged Sum Test	13	16
2	RGB Lagged Sum Test	14	16
2	RGB Lagged Sum Test	15	16
2	RGB Lagged Sum Test	16	17
2	RGB Lagged Sum Test	17	16
2	RGB Lagged Sum Test	18	19
2	RGB Lagged Sum Test	19	19
2	RGB Lagged Sum Test	20	19
2	RGB Lagged Sum Test	21	21
2	RGB Lagged Sum Test	22	19
2	RGB Lagged Sum Test	23	23
2	RGB Lagged Sum Test	24	25
2	RGB Lagged Sum Test	25	28
2	RGB Lagged Sum Test	26	29
2	RGB Lagged Sum Test	27	26
2	RGB Lagged Sum Test	28	23
2	RGB Lagged Sum Test	29	24
2	RGB Lagged Sum Test	30	25
2	RGB Lagged Sum Test	31	25
2	RGB Lagged Sum Test	32	27
2	RGB Kolmogorov-Smirnov Test	0	1
2	DAB Byte Distribution	0	1
2	DAB DCT	256	1
2	DAB Fill Tree Test	32	3
2	DAB Fill Tree 2 Test	0	4
2	DAB Monobit 2 Test	12	3
	Total		446

A.6 Static and Dynamic Scheduling Run-time

This section contains the raw experimental data that was collected for 4-thread static and dynamic scheduling.

Table A.14. 4-Thread Static Scheduling Tests Run-time - Thread 0

Thread ID	Statistical Test	ntuple	Run-time (Seconds)
0	Diehard Birthday Test	0	1
0	Diehard OPERM5 Test	0	4
0	Diehard 32x32 Binary Rank Test	0	14
0	Diehard 6x8 Binary Rank Test	0	2
0	Diehard Bitstream Test	0	1
0	Diehard OPSO Test	0	3
0	Diehard OQSO Test	0	2
0	Diehard DNA Test	0	29
0	Diehard Count the 1's (stream) Test	0	1
0	Diehard Count the 1's (byte) Test	0	2
0	Diehard Parking Lot Test	0	2
0	Diehard Minimum Distance (2d Circle) Test	2	1
0	Diehard 3d Sphere (Minimum Distance) Test	3	4
0	Diehard Squeeze Test	0	2
0	Diehard Sums Test	0	1
0	Diehard Runs Test	0	1
0	Diehard Craps Test	0	4
0	Marsaglia and Tsang GCD Test	0	161
0	STS Monobit Test	1	1
0	STS Runs Test	2	10
	Total		287

Table A.15. 4-Thread Static Scheduling Tests Run-time - Thread 1

Thread ID	Statistical Test	ntuple	Run-time (Seconds)
1	STS Serial Test	1	11
1	RGB Bit Distribution Test	1	2
1	RGB Bit Distribution Test	2	2
1	RGB Bit Distribution Test	3	2
1	RGB Bit Distribution Test	4	3
1	RGB Bit Distribution Test	5	3
1	RGB Bit Distribution Test	6	5
1	RGB Bit Distribution Test	7	7
1	RGB Bit Distribution Test	8	10
1	RGB Bit Distribution Test	9	14
1	RGB Bit Distribution Test	10	18
1	RGB Bit Distribution Test	11	26
1	RGB Bit Distribution Test	12	46
1	RGB Generalized Minimum Distance Test	2	6
1	RGB Generalized Minimum Distance Test	3	9
1	RGB Generalized Minimum Distance Test	4	24
1	RGB Generalized Minimum Distance Test	5	51
1	RGB Permutations Test	2	1
1	RGB Permutations Test	3	1
1	RGB Permutations Test	4	2
	Total		348

Table A.16. 4-Thread Static Scheduling Tests Run-time - Thread 2

Thread ID	Statistical Test	ntuple	Run-time (Seconds)
2	RGB Permutations Test	5	5
2	RGB Lagged Sum Test	0	1
2	RGB Lagged Sum Test	1	2
2	RGB Lagged Sum Test	2	3
2	RGB Lagged Sum Test	3	4
2	RGB Lagged Sum Test	4	5
2	RGB Lagged Sum Test	5	5
2	RGB Lagged Sum Test	6	7
2	RGB Lagged Sum Test	7	7
2	RGB Lagged Sum Test	8	8
2	RGB Lagged Sum Test	9	8
2	RGB Lagged Sum Test	10	9
2	RGB Lagged Sum Test	11	11
2	RGB Lagged Sum Test	12	11
2	RGB Lagged Sum Test	13	11
2	RGB Lagged Sum Test	14	12
2	RGB Lagged Sum Test	15	14
2	RGB Lagged Sum Test	16	14
2	RGB Lagged Sum Test	17	15
2	RGB Lagged Sum Test	18	16
	Total		242

Table A.17. 4-Thread Static Scheduling Tests Run-time - Thread 3

Thread ID	**Statistical Test**	**ntuple**	**Run-time (Seconds)**
3	RGB Lagged Sum Test	19	17
3	RGB Lagged Sum Test	20	17
3	RGB Lagged Sum Test	21	18
3	RGB Lagged Sum Test	22	19
3	RGB Lagged Sum Test	23	19
3	RGB Lagged Sum Test	24	20
3	RGB Lagged Sum Test	25	21
3	RGB Lagged Sum Test	26	22
3	RGB Lagged Sum Test	27	23
3	RGB Lagged Sum Test	28	23
3	RGB Lagged Sum Test	29	24
3	RGB Lagged Sum Test	30	25
3	RGB Lagged Sum Test	31	25
3	RGB Lagged Sum Test	32	26
3	RGB Kolmogorov-Smirnov Test	0	1
3	DAB Byte Distribution	0	2
3	DAB DCT	256	1
3	DAB Fill Tree Test	32	2
3	DAB Fill Tree 2 Test	0	4
3	DAB Monobit 2 Test	12	3
	Total		392

Table A.18. 4-Thread Dynamic Scheduling Tests Run-time - Thread 0

Thread ID	**Statistical Test**	**ntuple**	**Run-time (Seconds)**
0	Diehard Birthday Test	0	4
0	Diehard OPSO Test	0	3
0	Diehard DNA Test	0	26
0	RGB Bit Distribution Test	1	13
0	RGB Bit Distribution Test	5	18
0	RGB Bit Distribution Test	8	22
0	RGB Bit Distribution Test	11	46
0	RGB Generalized Minimum Distance Test	4	47
0	RGB Lagged Sum Test	0	1
0	RGB Lagged Sum Test	1	2
0	RGB Lagged Sum Test	3	6
0	RGB Lagged Sum Test	5	8
0	RGB Lagged Sum Test	8	11
0	RGB Lagged Sum Test	11	15
0	RGB Lagged Sum Test	14	17
0	RGB Lagged Sum Test	18	23
0	RGB Lagged Sum Test	22	29
0	RGB Lagged Sum Test	26	32
0	RGB Lagged Sum Test	30	36
0	DAB Fill Tree 2 Test	0	5
	Total		364

Table A.19. 4-Thread Dynamic Scheduling Tests Run-time - Thread 1

Thread ID	**Statistical Test**	**ntuple**	**Run-time (Seconds)**
1	Diehard 32x32 Binary Rank Test	0	18
1	STS Monobit Test	1	1
1	STS Runs Test	2	14
1	RGB Bit Distribution Test	2	10
1	RGB Bit Distribution Test	4	13
1	RGB Bit Distribution Test	6	18
1	RGB Bit Distribution Test	9	23
1	RGB Bit Distribution Test	12	72
1	RGB Permutations Test	2	1
1	RGB Permutations Test	3	1
1	RGB Permutations Test	4	3
1	RGB Permutations Test	5	7
1	RGB Lagged Sum Test	2	5
1	RGB Lagged Sum Test	4	7
1	RGB Lagged Sum Test	6	10
1	RGB Lagged Sum Test	9	13
1	RGB Lagged Sum Test	12	16
1	RGB Lagged Sum Test	15	21
1	RGB Lagged Sum Test	19	24
1	RGB Lagged Sum Test	23	28
1	RGB Lagged Sum Test	27	34
1	RGB Lagged Sum Test	31	34
	Total		373

Table A.20. 4-Thread Dynamic Scheduling Tests Run-time - Thread 2

Thread ID	**Statistical Test**	**ntuple**	**Run-time (Seconds)**
2	Diehard OPERM5 Test	0	8
2	Diehard Count the 1's (stream) Test	0	1
2	Diehard Count the 1's (byte) Test	0	2
2	Diehard Minimum Distance (2d Circle) Test	2	1
2	Diehard Squeeze Test	0	3
2	Diehard Sums Test	0	1
2	Diehard Runs Test	0	1
2	Diehard Craps Test	0	5
2	STS Serial Test	1	19
2	RGB Bit Distribution Test	3	18
2	RGB Bit Distribution Test	7	21
2	RGB Bit Distribution Test	10	31
2	RGB Generalized Minimum Distance Test	2	9
2	RGB Generalized Minimum Distance Test	3	17
2	RGB Generalized Minimum Distance Test	5	101
2	RGB Lagged Sum Test	17	22
2	RGB Lagged Sum Test	21	28
2	RGB Lagged Sum Test	25	31
2	RGB Lagged Sum Test	29	35
2	RGB Kolmogorov-Smirnov Test	0	2
2	DAB Byte Distribution	0	2
2	DAB DCT	256	2
2	DAB Fill Tree Test	32	4
2	DAB Monobit 2 Test	12	4
	Total		368

Table A.21. 4-Thread Dynamic Scheduling Tests Run-time - Thread 3

Thread ID	**Statistical Test**	**ntuple**	**Run-time (Seconds)**
3	Diehard 6x8 Binary Rank Test	0	4
3	Diehard Bitstream Test	0	1
3	Diehard OQSO Test	0	4
3	Diehard Parking Lot Test	0	2
3	Diehard 3d Sphere (Minimum Distance) Test	3	4
3	Marsaglia and Tsang GCD Test	0	179
3	RGB Lagged Sum Test	7	10
3	RGB Lagged Sum Test	10	14
3	RGB Lagged Sum Test	13	17
3	RGB Lagged Sum Test	16	23
3	RGB Lagged Sum Test	20	25
3	RGB Lagged Sum Test	24	29
3	RGB Lagged Sum Test	28	34
3	RGB Lagged Sum Test	32	33
	Total		379

List of References

[1] Popular “encrypted chat” service cryptocat contained a vulnerability for 7 months. (2013, Jul. 18). Department of Communications and the Arts. [Online]. Available: https://www.communications.gov.au/what-we-do/internet/stay-smart-online/alert-service/popular-encrypted-chat-service-cryptocat-contained-vulnerability-7-months

[2] D. Goodin. (2013, Jul. 5). Bad kitty! “Rookie mistake” in cryptocat chat app makes cracking a snap. Ars Technica. [Online]. Available: http://arstechnica.com/security/2013/07/bad-kitty-rooky-mistake-in-cryptocat-chat-app-makes-cracking-a-snap/

[3] DSA-1571-1 Openssl – Predictable random number generator. (2008, May 13). Debian Security Advisory. [Online]. Available: https://www.debian.org/security/2008/dsa-1571

[4] Android crypto PRNG flaw aided bitcoin thieves, Google releases patch. (2013, Aug. 16). SiliconANGLE. [Online]. Available: http://siliconangle.com/blog/2013/08/16/android-crypto-prng-flaw-aided-bitcoin-thieves-google-releases-patch/

[5] M. Gondree. (2014, Sep. 28). NPS POSIX thread pool library. [Online]. Available: https://github.com/gondree/nps-tpool

[6] Frequently asked questions on OpenMP. (n.d.). OpenMP ARB Corporation. [Online]. Available: http://openmp.org/openmp-faq.html. Accessed Jul. 1, 2016.

[7] A. Desai, A. Hevia, and Y. L. Yin, *A Practice-Oriented Treatment of Pseudorandom Number Generators*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 368–383. Available: http://dx.doi.org/10.1007/3-540-46035-7_24

[8] P. L’Ecuyer, “Random numbers for simulation,” *Commun. ACM*, vol. 33, no. 10, pp. 85–97, Oct. 1990. Available: http://doi.acm.org/10.1145/84537.84555

[9] M. Blum and S. Micali, “How to generate cryptographically strong sequences of pseudorandom bits,” *SIAM Journal on Computing*, vol. 13, no. 4, pp. 850–864, 1984. Available: http://dx.doi.org/10.1137/0213053

[10] A. C. Yao, “Theory and application of trapdoor functions,” in *Foundations of Computer Science, 1982. SFCS ’08. 23rd Annual Symposium on*, Nov 1982, pp. 80–91.

[11] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo, “A statistical test suite for random and pseudorandom number generators for cryptographic applications,” National

Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST SP 800-22, May 2001.

[12] P. L'Ecuyer and R. Simard, "Testu01: A c library for empirical testing of random number generators," *ACM Trans. Math. Softw.*, vol. 33, no. 4, Aug. 2007. Available: http://doi.acm.org/10.1145/1268776.1268777

[13] G. Marsaglia. (n.d.). Diehard test descriptions. [Online]. Available: http://diehard/cdrom/dos/tests.txt. Accessed Apr. 1, 2016.

[14] G. Marsaglia. (n.d.). Diehard test suite readme. [Online]. Available: http://stat.fsu.edu/pub/diehard/cdrom/dos/diehard.doc. Accessed Apr. 1, 2016.

[15] R. G. Brown. (n.d.). Diehard: A random number test suite. Duke University Physics Department. Durham, NC 27708-0305. [Online]. Available: https://www.phy.duke.edu/~rgb/General/dieharder.php. Accessed Jun. 1, 2016.

[16] R. G. Brown. (2011, Oct. 14). List of current fully implemented tests. [Online]. Available: http://www.phy.duke.edu/~rgb/General/dieharder/dieharder-3.31.1.tgz

[17] Ubuntu manpage: dieharder - A testing and benchmarking tool for random number. (n.d.). Canonical Ltd. [Online]. Available: http://manpages.ubuntu.com/manpages/trusty/man1/dieharder.1.html. Accessed Jul. 1, 2016.

[18] What is Amazon EC2? (n.d.). Amazon. [Online]. Available: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html. Accessed Jul. 1, 2016.

[19] Amazon EC2 instance types. (n.d.). Amazon. [Online]. Available: https://aws.amazon.com/ec2/instance-types/. Accessed Jul. 1, 2016.

[20] Ubuntu manpage: time - Run programs and summarize system resource usage. (n.d.). Canonical Ltd. [Online]. Available: http://manpages.ubuntu.com/manpages/trusty/man1/time.1.html. Accessed Jul. 1, 2016.

[21] Amazon EBS volume types. (n.d.). Amazon. [Online]. Available: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html. Accessed Jul. 1, 2016.

[22] Ubuntu manpage: random, urandom - Kernel random number source devices. (n.d.). Canonical Ltd. [Online]. Available: http://manpages.ubuntu.com/manpages/trusty/man4/random.4.html. Accessed Jul. 1, 2016.

[23] T. Huhn. (2015, Apr. 11). Myths about /dev/urandom. [Online]. Available: http://www.2uo.de/myths-about-urandom/

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California